

Automatic Back Annotation of Timing into VHDL Behavioral Models

Gayatri Mahadevan and James R. Armstrong
Department of Electrical Engineering
Virginia Tech

Abstract

This paper presents a design system that significantly speeds up development of VHDL behavioral models with back annotated timing. The behavioral model is developed using the CAD tool called Modeler's Assistant by inputting the model in the form of a Process Model Graph. The back annotation tool Backann2 uses the VHDL description from the Modeler's Assistant, the Compass-VTIP CAD tool and the Synopsys Design Compiler to calculate the timing delays and to back annotate the delays into the behavioral model. The Compass-VTIP tool is used to extract the details from the VHDL model and store it in the form of data structures. These details are used for computing the paths traversed by the signals associated with the generics. The behavioral model is synthesized into a gate level design and the end to end delays in the model are obtained using Synopsys Design Compiler. An algorithm has been developed which, given the end to end delays and the different paths traversed by the signals, finds realistic and accurate delays for processes along the path. Thus a system is available to designers which builds behavioral models with accurate timing information.

1. Introduction

A simulation that performs an accurate timing analysis can give the designer the leeway to test the timing tolerance of the design by simulating it for different conditions [1]. With early HDL's the behavioral models were simulated to verify the behavior and timing was not considered.

The accuracy of the simulation of a VHDL behavioral model created using generics depends upon the accuracy of the values specified for the generics. The design of a complex system becomes a time consuming and labor intensive task if the designer has to calculate these delays manually. It is even more difficult to calculate these values without a physical representation of the chip. Also, the actual delay values in the chip may vary from what the designer calculated and incorporated for the model's simulation. This might lead to race problems and timing hazards [2, 3] at a later stage in the design process. With the synthesis tools available now the behavioral model can be synthesized into a gate level circuit and the delays can be obtained.

In a behavioral model that is comprised of processes, each processes is associated with delays. Generics are used to represent these delays associated with each process. Thus the problem is to supply the model with accurate generic values for it to simulate correctly [5, 6]. This is one of the obstacles facing every designer.

2. VHDL Behavioral Model

The VHDL description of a system is a textual representation. Development of a VHDL behavioral model using text alone is a time consuming process. The model development may be speeded up by using pictorial representations combined with text. The pictorial representations make it easier to specify the interconnections between the different components in the system. Model development is provided using a

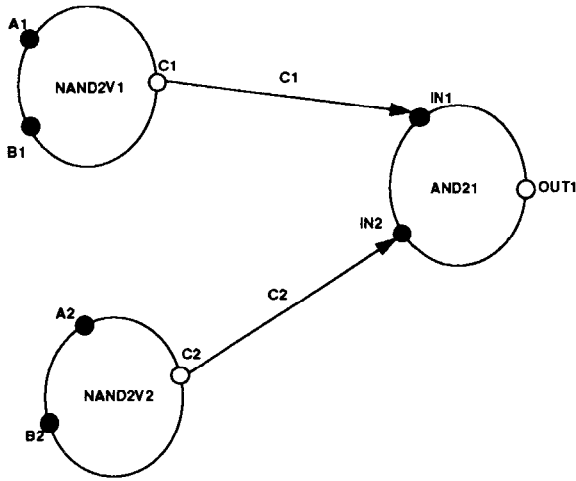


Figure 1 Process Model Graph generated using Modeler's Assistant

program called the Modeler's Assistant [8, 9]. It employs a graphical representation of a VHDL behavioral model called a Process Model Graph (PMG). The nodes are processes and the arcs are signals between the processes. A PMG is used to represent the division of functionality within a VHDL behavioral model and gives an interconnection of different processes that partition the model into interrelated but distinct functional modules. Figure 1 shows a Process Model Graph generated by Modeler's Assistant and the VHDL representation generated by it is shown in Figure 2.

3. Earlier Work

For designers, at any of the levels of hierarchy, one of the tasks facing them is the analysis of timing. In the past timing analysis was performed only at the gate level. This is because the design was not modeled at higher levels. With the advent of HDL's it is possible to model the designs at an abstract level and perform timing analysis. It is an advantage to do timing at higher levels to verify the interaction of the different modules. If timing is not considered at any level, erratic behavior of the system, resulting in hazards may occur during actual implementation of the system. Various algorithms are being investigated to overcome this problem at different levels of the design

```

-- *****
entity EG1 is
  generic (
    AND_DEL: TIME ;
    NAND2_DEL2: TIME ;
    NAND2_DEL1: TIME
  );
  port (OUT1: out BIT;
        B2: in BIT;
        A2: in BIT;
        B1: in BIT;
        A1: in BIT);
end EG1;
-- *****

architecture BEHAVIORAL of EG1 is
  signal C2: BIT;
  signal C1: BIT;
begin
  -----
  -- Process Name: AND21
  -----
  AND21_4: process (C2,C1)
  begin
    OUT1 <= C1 and C2 after AND_DEL ;
  end process AND21_4;
  -----
  -- Process Name: NAND2V2
  -----
  NAND2V2_10: process (B2,A2)
  begin
    C2 <= not(A2 and B2) after NAND2_DEL2 ;
  end process NAND2V2_10;
  -----
  -- Process Name: NAND2V1
  -----
  NAND2V1_16: process (B1,A1)
  begin
    C1 <= not(A1 and B1) after NAND2_DEL1 ;
  end process NAND2V1_16;

end BEHAVIORAL;

```

Figure 2 VHDL description of the above Process Model Graph

hierarchy [4].

A. Gadagkar and J.R. Armstrong [5, 6] developed an algorithm for timing distribution inside each process for a given end to end timing. The methodology detects and corrects

inconsistencies in the timing specifications and allocates block delays to meet the specifications. This has been incorporated in a tool called TIMESPEC. A weakness of this tool is that the delays calculated may not be realistic.

S. Narayanaswamy and J.R. Armstrong [7] developed an algorithm Backann to find the individual process and signal assignment delays. Backann determines the delay values that are required for the signal assignments in the behavioral model by generating the gate-level design of the model using the Synopsys Design Compiler. It extracts the values for the delays required from the gate-level design. It then back annotates these values into the VHDL behavioral model. The VHDL model is in the form of a PMG. Backann extracts each process from the PMG, converts it into a separate entity, synthesizes it, extracts the signal assignment delays and back annotates them into the VHDL model. Though it calculates the signal assignment delays for each signal accurately, it does not take into fact that when the whole model is synthesized, the design obtained will not be the same as the one where each process is synthesized individually. This is because when synthesizing a gate-level design, logic from different processes is frequently merged and optimized.

The approach used in backann can be modified and refined to calculate accurate delays for the whole synthesized model. Below we describe this modified approach in detail.

4. Algorithm Development

The algorithm developed consists of three steps: Path enumeration, synthesis using the Synopsys Design Compiler and calculation of delays.

4.1 Path Enumeration

When assigning exact delay values to the generics that are provided as a part of the PMG description, one must consider the path traversed by the signals associated with the generics. This can be achieved by identifying all

the paths from the primary inputs to the primary outputs in the PMG and enumerating them in terms of the generic delays along the paths traversed. For example, the paths from the primary inputs A1 (B1) and A2 (B2) to the primary output OUT1, for the PMG shown in Figure 1, are defined in terms of the processes generics as follows:

$$\begin{aligned} \text{PATH1} &= \text{NAND2_DEL2} + \text{AND_DEL} \\ \text{PATH2} &= \text{NAND2_DEL1} + \text{AND_DEL} \end{aligned}$$

where NAND2_DEL2, NAND2_DEL1 and AND_DEL are the generic delays associated with processes NAND2V2, NAND2V1 and AND21 respectively.

To extract the details from the VHDL code of the model for path enumeration, the Compass-VTIP CAD tool is used.

4.1.2 Compass VTIP, DLS and SPI

As mentioned above, the Compass-VTIP CAD tool is used to extract the details from the VHDL model and store it in the form of data structures. In this application, the input to VTIP is the VHDL code obtained from Modeler's Assistant. The analyzer processes the VHDL description, checks for syntactic and semantic errors and stores it in an internal format in the Design Library System (DLS) [10, 11].

The DLS consists of two fundamental components - one abstract, the other concrete. The abstract component is the DLS Data Definition, which specifies the data elements, objects and structures that are supported by the DLS, together with the abstract operations that can be performed upon them. The concrete component is the Software Procedural Interface (SPI), which is a software implementation of the DLS Data Definition that supports the development of applications based on the DLS Data Definition.

The DLS Data Definition consists of three layers: the Data Model, the Schema and the Information Model. The Data Model defines the basic data elements (e.g., BOOLEAN, INTEGER, etc.) that can be used to represent

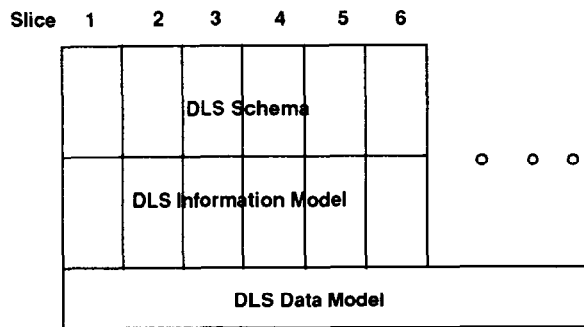


Figure 3 Structure of DLS Data Definition

the simplest concepts in any design data (such as numbers, strings, lists, etc.). Some of these elements are *generics** (like NODES, ATTRIBUTES, LISTS, ITEMS). The Schema defines specific versions of such generic objects, and in doing so specifies their interpretation. The Information Model defines how the specific objects can be assembled into a complex structure. It mainly describes how libraries and library units store design data information in a domain with a hierarchy of region nodes forming the backbone of the domain.

The SPI is the fundamental part of the DLS [12]. The SPI consists of data types and callable routines that implement the data types and operations of the DLS. The SPI provides layers of data types and routines to support the layers (primitive and generic data defined by the DLS model) of the DLS Data Definition. The SPI is implemented in the C programming language. These routines provide high level functions that can be implemented in terms of lower-level functions. They support addition of extra information to nodes, evaluation of expressions, symbol searches and generalized queries of DLS structures. These SPI routines are made accessible by compiling the 'C' source files with the SPI library functions.

With the help of the SPI functions the paths from the VHDL model's inputs to the outputs are obtained. It is easier to visualize the VHDL code that was parsed and stored in the DLS as a

* as defined by the VTIP CAD tool, not the VHDL definition

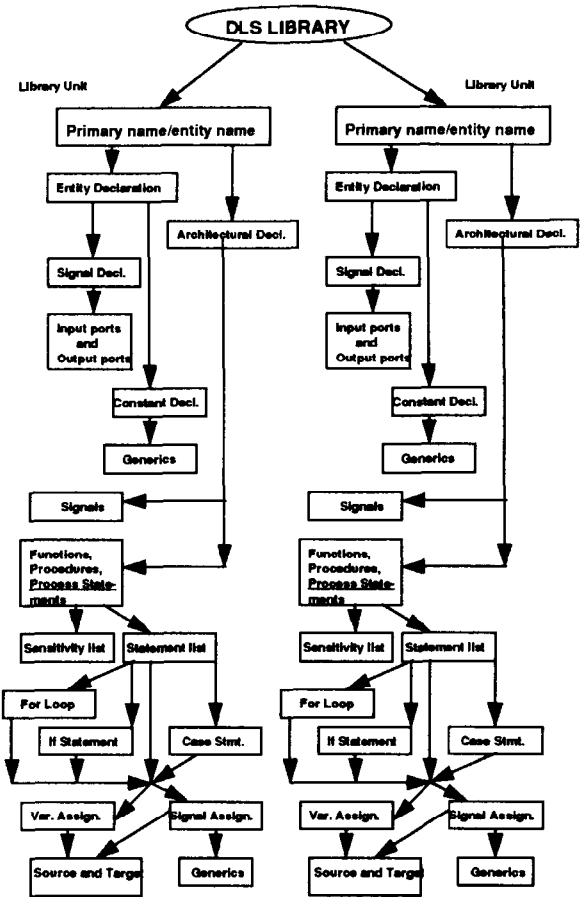


Figure 4 The VTIP Design Library System (DLS)

tree of data structures whose top starts with a string that specifies the name of the file in which the description is stored. This is called the main unit or the library unit. The tree then moves to what is called the body where the entity and the architecture declarations are made. In the entity declaration branch the input and output ports and the generic values are all stored. In the architecture declaration branch the different processes, procedures, function declarations or statements are stored if it is a behavioral model, and component instantiations are stored in case of structural models. The branches keep on growing till all the subtypes (statements) are covered. This is shown in Figure 4 (for behavioral models).

SPI functions are called by the program to extract information. The information extracted

```

Open the DLS library unit
Access the architectural body (secondary name) of the unit
Obtain the different process statements.
For Each Process,
    Assign a unique number to it (process-number).
    To get all the inputs scan the sensitivity list of the process.
    Obtain all the inputs to the process
        for each input,
            Get the mode ( IN -> primary input,
                          INOUT-> an intermediate signal)
    To get each output in the process scan all the signal assignment statements.
    Get all the outputs of the process
        for each output,
            Get the mode (OUT-> primary output,
                          INOUT -> an intermediate signal)
            Get the delay associated with each output.
    Store these in the form of data structures
Close the DLS unit

```

Figure 5 Algorithm for obtaining the data from DLS library

using these functions consists of process names, the inputs to each process, the outputs of each process, the mode of the input and output signals and the generic delay associated with each signal assignment inside each process. Each process is given a unique number for identification. According to VTIP, a signal can be classified into one of the three modes. If a signal is an output or input to the whole model then it is of mode OUT or IN respectively, else the signal is of mode INOUT* [10, 11]. These are stored with other details, as shown in Figure 5, in the form of data structures.

The signal paths in the VHDL model are obtained by implementing the Depth First Search (DFS) Algorithm [16]. As mentioned earlier, each of the processes in the PMG is considered to be a node and the sensitivity list to each process is considered as inputs to the node. To obtain a path the algorithm visits every node in the graph, the first node and the last node being the process that contains a primary input and the process that contains a primary output respectively, and checks every edge in the graph systematically. The edges for

the processes are checked by inspecting the adjacency list.

4.1.3 Depth First Search Algorithm

In order to obtain all the paths in the VHDL model the Depth First Search Algorithm[16] was modified and used. To implement this algorithm the connectivity details of the processes in the model should be in the form of an adjacency list. This is obtained from an adjacency matrix. The algorithm for obtaining the adjacency matrix and the adjacency list is shown in Figure 6. For the PMG shown in Figure 1, the adjacency matrix and the adjacency list are shown below.

Adjacency matrix :

0	0	0
1	0	0
1	0	0

Adjacency list :

```

for process 1 : 1
for process 2 : 2      --> 1
for process 3 : 3      --> 1

```

* as defined by the VTIP CAD tool, not the VHDL definition

```

Create an NxN matrix (N -- number of processes in the model). Make all its entries 0

For all the processes,
    Compare the outputs of each process with the inputs of every other process.
    If the two signals match (and the mode of these signals is INOUT) then the
    Output of the former process is connected to the input of the later process.
    If the above condition is true then set the matrix entry to 1.
The matrix thus formed is the Adjacency matrix.
In the adjacency matrix,
    For each row in the matrix
        for each column in that row,
            Check if the entry is 1,
            if it is 1 then push the previous node value and store the new
            value in the list
            else continue till all the columns are scanned.

Perform the same operation for each row and obtain the list for each row

```

Figure 6 Algorithm for obtaining the adjacency matrix and the adjacency list

The program is written in such a way that an array is filled (mark_val[]) as it visits every node of the graph. The array is initially set to all zeros, so that the entry for mark_val for the i th node is a one when visited. To visit a node, we check all the edges to and from the node to see if they lead to nodes that haven't yet been visited (as indicated by zero values); if so visit them. The above mentioned recursive function, which visits all the intermediate nodes from the primary input to the primary output, is called and all paths for a particular combination of primary input to primary output is obtained. The output of each process connects to the input of another process. Once the primary output node is reached the algorithm traces back to the primary input node and obtains all the paths. The same procedure is used to trace all the paths from different primary inputs to primary outputs. With the data obtained from the DLS and the adjacency list the paths are obtained using this algorithm. The algorithm is explained in Figure 7. It generates the paths in the form of node numbers. For example consider Figure 1, its adjacency matrix and adjacency list. The primary inputs to the model are the inputs to Process 2 and Process 3. The primary output of the model is the output of Process 1. Using Depth First Search algorithm recursively, we obtain

2 1 and 3 1
as the paths. Each path is stored in a linked list. The paths obtained in the form of node numbers are transformed to the form of generics. The algorithm for doing this is given in Figure 8.

4.2 Synthesis of the Model Using the Synopsys Design Compiler

The Synopsys Design Compiler is invoked from the program through the UNIX Bourne shell, and after synthesizing the design, gives an optimized gate level circuit. The level of optimization and the constraints for the design should be given before optimizing. Since the circuit is optimized the intermediate signals cannot be identified unless the design and the reports for the design are analyzed manually. Only the input and output ports are visible to the designer. The set of commands for reading in and then optimizing the design is done by creating a script file that will perform all the functions required by the user. This file is specified in the command line to the Synopsys Design Compiler and the required tasks are performed [13].

1. Obtain the start and end node (this is the process number of the nodes with primary inputs and primary outputs)
 2. Create an array and initialize all its elements to zero (*val_mark[]*). This array is to denote whether the node has been visited.
 3. Create an array which contains the latest path (*que[]*) and a variable (*last*) which gives the number of entries in the array.
 4. If the start and end node is the same, then a path has been obtained else store the start node in a temp variable.
 5. Assign the link list to a variable of the same type.(say *p*)
 6. If the node, in the linked list points to another node say *x* and the value of *val_mark[x]* is equal to 0, add *x* to *que* and perform 4 to 6 again till the end node is reached.
 7. If the node, in the linked list points to another node say *x* and the value of *val_mark[x]* is equal to 1, *p* = linked field of *p*.
- Perform steps 1 to 7 for all start-end node combinations.

Figure 7 DFS Algorithm to find all the paths between inputs and outputs

1. Obtain the initial node (number) and store it in *i*
 2. Obtain the next node number from the linked list and store it in *j*
 3. Find the processes corresponding to *i* and *j*
 4. If *i* and *j* are equal then it is the final node, and so obtain the output of the process and the generic delay associated with it. To obtain the output the signal will be of mode OUT.
 5. If *i* is not equal to *j* compare the output signal of process *i* and input signal of process *j* and if they are equal and the mode is INOUT then find the generic associated with it.
Store the generic delays associated with each path in a file.
 6. Now *i* = *j* and *j* = next number in the path.
- Perform steps 4 -6 for all nodes in the path and steps 1 to 6 for all paths

Figure 8 Algorithm for representing the paths in terms of generic delays

The Synopsys Design Compiler is invoked in two parts of the algorithm. In the first part it is invoked by earlier version of *backann*, to synthesize each process in the model separately. *Backann* extracts [14] each process, stores it as a separate entity and uses the Synopsys Design Compiler to synthesize the gate level design and calculate the end to end delays of each process, by calculating the delay values for signal assignments in the process. *Backann* thus calculates each process delay and then stores it back into the VHDL model as generics. The designer can specify whether a minimum delay model (minimum rise and minimum fall) or a maximum delay model (maximum rise and

maximum fall) is required. Depending on the type of delay model required, (maximum or minimum) the generic delays are calculated as an average of rise and fall delays. The commands for executing the Design Compiler are obtained from a script file. The script file is contained in a log-file given by the user. A sample *log_file* is shown in Figure 9. The specifications to the designer include the design libraries to be used, the different components like specific flip-flop and latch types, maximum and minimum fanout and fanin. Constraints like area, speed, setup time, hold time can also be specified. The file also includes the level of optimization and verification.

```

search_path = ". /project/rassp1/synopsys3.2b/libraries/syn /project/rassp1/synopsys3.2b/libraries
/project/rassp1/synopsys3.2b/sparc/packages "
link_library = "lsi_10k.db "
target_library = "lsi_10k.db "
symbol_library = "lsi_10k.sdb "
default_schematic_options = "-size infinite"
hdlin_source_to_gates_mode = "off"
create_schematic -size infinite -gen_database
set_operating_conditions -library "lsi_10k" "BCCOM"
set_wire_load "20x20" -library "lsi_10k"
set_max_fanout 5 " /home/mahadev/scode/EG1.db:EG1"
set_min_fault_coverage 95 -area_critical -timing_critical
set_register_type -flip_flop "FJK2SP"
derive_timing_constraints -no_max_period -min_delay -fix_hold -max_delay_scale 1.00 -min_delay_scale
1.00 -period_scale 1.00
set_boundary_optimization " /home/mahadev/scode/EG1.db:EG1"
set_flatten true -design{"/home/mahadev/scode/EG1.db: EG1"} -effort high -minimize none -phase true
set_structure true -design{"/home/mahadev/scode/EG1.db: EG1"}-boolean true -timing true
link -all
compile -map_effort high -verify -verify_effort high -boundary_optimization
report_cell >> rep.out
report_area >> rep.out
report_constraints -all_violators >>rep.out
report_timing -from A1 -to OUT1 -max_paths 2 >>rep.out
report_timing -from B1 -to OUT1 -max_paths 2 >>rep.out
report_timing -from A2 -to OUT1 -max_paths 2 >>rep.out
report_timing -from B2 -to OUT1 -max_paths 2 >>rep.out

```

Figure 9 The logfile of user specified commands to the synthesizer for Figure 1

```

Open the report file
Open a new file to store the extracted data (TMPREP.OUT)
If string "Startpoint: " is found in the report file and
If string "Endpoint: " occurs in the next line,
then scan the next few lines for Arrival Time.
If arrival time is found then copy the value along with Startpoint and
Endpoint into the file TMPREP.OUT
Continue the procedure till all arrival times are obtained.

```

Figure 10 Algorithm for extracting the delay values from the report file

The main constraint used for calculating the generics for the given model was minimum area. If registers are used in the circuit, the D flip-flop was used. The maximum fanout of each gate in the circuit was restricted to five

and Boolean optimization was used. The Design Compiler is invoked by a script file created by the software. The first line of the script file is

```
read -f <vhdl_file_name_to be synthesized>
```

which when executed would read in the VHDL file to be synthesized. To open the Design Compiler the command used is

```
dc_shell -f <script_file_name>
```

The contents of the log_file created by the user is appended to the script file. The log_file also includes commands to extract delays in Synopsys Design Compiler. It is done by using *report_timing* [28]. The *report_timing* command not only reports the delay values required but also other information. The other information includes the technology used, the wire delay model, which determines the propagation delay of a signal through a wire between two cells, the start and end points of the signal, type of delay (maximum or minimum), the intermediate cells through which the signal is propagated and delays through each of the cells. These are all stored in the report file *rep.out*.

The program then scans the report file and obtains the delays and stores them in a file. The algorithm for scanning the report file *rep.out* is given in Figure 10. It scans for the string *data arrival time* in the report file and obtains the time taken for the signal. The output is stored in a file *TMPREP.OUT*.

4.3 Circuits and Their Types

Circuits can be broadly classified as combinational and sequential. A combinational circuit is one whose outputs depend only on its current inputs. The outputs of a sequential circuit depend not only on the current inputs but also on the past sequence of inputs, possibly far back in time. A feedback loop is a signal path of a circuit that allows the output of a gate to propagate back to the input of the same or another gate; such a loop generally creates a sequential circuit behavior.

In our application which uses Process Model Graphs, the above mentioned circuits can be classified as:

Class 1 : Combinational fanout free circuits.

Class 2 : Combinational circuits with fanout.

Class 3 : Register Sequential Circuits

Class 4 : Highly Sequential Circuits

Fanout free combinational circuits are circuits in which the fanout of the outputs of each process is one. In graphical terms this type of circuit is a binary tree, where the root of the tree is the output and the leaves are the inputs. Circuits in which the outputs of the processes are fed in as inputs to more than one process are classified as Class 2 circuits. Class 3 circuits are sequential circuits that alternate blocks of combinational logic with registers. Class 4 circuits are sequential circuits which either have feedback loops or irregular register or flip-flop structures. All the above circuits can either have asymmetrical nodes (e.g. : ADDER's, MUX's) or symmetrical nodes (e.g. : AND gates, NOR gates etc.).

The principle for calculating the generic delays for the first three classes is discussed. The circuits are subdivided into these classes for better understanding of the algorithm used to obtain the delays.

4.3.1 Calculation of Delays

The VHDL model illustrated in Figure 1 is a Class 1 circuit. Using the result obtained from Depth First Search Algorithm and the algorithm mentioned in Figure 7 the paths obtained from the primary inputs to the primary outputs are,

$$\begin{aligned} \text{PATH1} &= \text{NAND2_DEL2} + \text{AND_DEL}; \\ \text{PATH2} &= \text{NAND2_DEL1} + \text{AND_DEL}; \end{aligned}$$

When this model is synthesized as a whole, the functional paths that are considered are *PATH1* and *PATH2*. When the timing is reported we can find the time taken for a signal to traverse through *PATH1* and *PATH2*.

When each of these processes is converted into a separate entity (as in Backann), the optimization would be only for that process, but when the whole model is being optimized the gate level circuit of the design can no longer be identified as individual processes. The whole model would be optimized and redundancy in the code would also be checked for. The end to

When this model is synthesized as a whole, the functional paths that are considered are PATH1 and PATH2. When the timing is

1. Consider all the paths from input to output
2. Obtain the values of each path from the Synopsys Design Compiler
3. From backann obtain the individual generic delay.
4. If there are paths with only one generic, i.e. if it traverses only one process obtain the generic and store it. Also, overwrite the value earlier obtained for the generic.

$$\text{genericX} = \text{total delay of the path (single process delay)}$$

5. If in any of the paths, if any of the above generic value occurs, subtract the generic from the path and obtain the new path delay. This will be equal to the sum of the remaining generic delays, which are to be obtained.
6. If after performing step 5 any singular generic delay is obtained, steps 4 and 5 are repeated until all delays are exhausted.
7. If either all the singular delays are exhausted and the paths are equivalent to sum of individual generic delays or steps 4,5 and 6 were never performed, consider the path with minimum number of generics. For each generic in that path,

$$\text{genericY}_{\text{new}} = \text{genericY} / \langle \text{new_path_delay} \rangle * \text{path_delay} \quad (1)$$

$$\text{genericY} = \text{genericY}_{\text{new}} \quad (2)$$

where genericY in (1) is the value calculated from backann, path delay - is the sum of individual delays obtained from Synopsys Design Compiler or from step 5 and new_path_delay is the sum of the generics in that path obtained from backann. The path maybe either the entire path from input to output or a portion of the path if Step 5 is performed.

8. After step 7 check if any of the remaining paths has a generic just calculated. If it does perform step 5 and 6 and if necessary steps 7 and 8 till all delays are exhausted. If it does not then perform steps 7 and 8.

Figure 11 Algorithm for calculating the delays

end delays will in general be different than the sum of individual process delays. They may be more or less than the original values of PATH1 or PATH2 depending on the constraints applied during synthesis.

The goal is to find individual process delays which would satisfy the given constraints when the model is synthesized. It is clear that when the end to end path delay changes, the logic corresponding to the processes in the original model also changes accordingly. This means that an increase or decrease in the end to end delay signifies a similar change in the original process delays. The word original is used because the gate level circuit of the VHDL model will perform the same function as the behavioral model, but the similarity ends there.

The problem of finding the generic delays reduces to a linear programming ratio problem. The algorithm for finding the individual process delays based on this concept is shown in Figure 11. Consider the above mentioned example. The new generic delays from the algorithm (step 4 and step 5) will be of the following form,

$$\text{NAND2_DEL2}_{\text{new}} = \text{NAND2_DEL2} / \text{PATH1} * \langle \text{path delay from design comp} \rangle \dots (1)$$

$$\text{NAND2_DEL2} = \text{NAND2_DEL2}_{\text{new}} \dots (2)$$

$$\text{AND_DEL}_{\text{new}} = \text{AND_DEL} / \text{PATH1} * \langle \text{path delay from design comp} \rangle \dots (3)$$

$$\text{AND_DEL} = \text{AND_DEL}_{\text{new}} \dots (4)$$

$$\text{NAND2_DEL1}_{\text{new}_2} = \langle \text{path delay from design comp} \rangle - \text{AND_DEL} \dots (5)$$

The new delay values obtained are shown below. They are back-annotated into the VHDL model. A comparison of these values with actual gate level values from the synthesis report is shown in Figure 13.

<i>Generic delays of the model</i>	<i>Values from the report</i>	<i>Values from Backann2</i>
OR2_DEL	0.3600	0.3600
INVSECDEL2	0.4900	0.4900
INVVDELA	0.1850	0.1850
INVVDEL1	0.1800	0.1819
AND3DEL1	0.5400	0.5336

Figure 13 Comparison of Delays

OR2_DEL	=	0.3600
INVSECDEL2	=	0.4900
INVVDELA	=	0.1850
AND3DEL1	=	0.5336
INVVDEL1	=	0.1819

It is seen that the error percentage for INVVDEL1 and AND3DEL1 is 1.05% and 1.18% respectively. The other generics have zero error. The average error percentage is 1.12%.

5.2 ADDER-MUX circuit

This circuit is an example of a fanout circuit. The nodes in this circuit are asymmetrical. The Process Model Graph in Figure 14 shows two multiplexers, two adders and five processes with simple combinational logic. The different paths in the circuit are illustrated below.

$TP1 = MUPX2_DEL + SUM2_DEL + OR2S_DEL$
 $TP2 = SUM2_DEL + OR2S_DEL$
 $TP3 = ENABLE_DEL + SUM2_DEL + OR2S_DEL$
 $TP4 = OR2S_DEL$
 $TP5 = FN1_DEL$
 $TP6 = ENBL_DEL + FN1_DEL$
 $TP7 = MUPX_DEL + FN1_DEL$
 $TP8 = MUPX2_DEL + CARRY2_DEL$
 $TP9 = ENABLE_DEL + CARRY2_DEL$
 $TP10 = MUPX2_DEL + SUM1_DEL$
 $TP11 = MUPX2_DEL + CARRY1_DEL$
 $TP12 = MUPX_DEL + SUM1_DEL$

$TP13 = MUPX_DEL + CARRY1_DEL$
 $TP14 = ENABLE_DEL + SUM1_DEL$
 $TP15 = ENABLE_DEL + CARRY1_DEL$
 $TP16 = MUPX_DEL + AND_DEL$
 $TP17 = AND_DEL$

The equations for calculating the generic delays and the results obtained from MATLAB for these equations are shown below.

$FN1_DEL = 0.770000$
$ENBL_DEL = 0.985000$
$MUPX2_DEL = 1.320000$
$MUPX_DEL = 1.285000$
$CARRY2_DEL = 1.405000$
$SUM2_DEL = 2.375000$
$CARRY1_DEL = 1.405000$
$SUM1_DEL = 2.270000$
$ENABLE_DEL = 1.820000$
$AND_DEL = 0.625000$
$OR2S_DEL = 0.615000$

$TP1 = 3.77$
 $TP2 = 2.66$
 $TP3 = 5.33$
 $TP4 = 0.38$
 $TP5 = 0.69$
 $TP6 = 1.67$
 $TP7 = 1.76$
 $TP8 = 2.24$
 $TP9 = 3.80$
 $TP10 = 3.25$
 $TP11 = 2.24$
 $TP12 = 3.20$
 $TP13 = 2.20$
 $TP14 = 4.80$
 $TP15 = 3.80$
 $TP16 = 1.55$
 $TP17 = 0.48$
 $TP17 = 0.48$

$FN1_DEL = 0.625000$
 $ENBL_DEL = 0.755000$
 $MUPX2_DEL = 0.690000$
 $MUPX_DEL = 0.690000$
 $CARRY2_DEL = 0.855000$
 $SUM2_DEL = 1.315000$
 $CARRY1_DEL = 0.855000$
 $SUM1_DEL = 1.315000$
 $ENABLE_DEL = 0.460000$
 $AND_DEL = 0.360000$

OR2S_DEL = TP4
 FN1_DEL = TP5
 AND_DEL = TP17

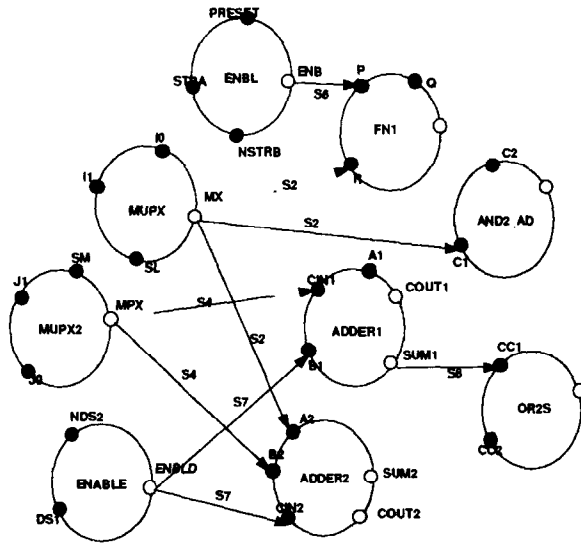


Figure 14 Process Model Graph of the ADDER-MUX circuit

SUM2_DEL = TP2-OR2S_DEL
 ENBL_DEL = TP6-FN1_DEL
 MUPX_DEL = TP7-FN1_DEL
 MUPX_DEL = TP16-AND_DEL
 MUPX2_DEL = TP1-OR2S_DEL-SUM2_DEL
 ENABLE_DEL = TP3-OR2S_DEL-SUM2_DEL
 SUM1_DEL = TP12-MUPX_DEL
 CARRY1_DEL = TP13-MUPX_DEL
 CARRY2_DEL = TP8-MUPX2_DEL
 CARRY2_DEL = TP9-ENABLE_DEL
 SUM1_DEL = TP10-MUPX2_DEL
 CARRY1_DEL = TP11-MUPX2_DEL
 SUM1_DEL = TP14-ENABLE_DEL
 CARRY1_DEL = TP15-ENABLE_DEL

The generics obtained from backann2 are compared to the corresponding values obtained from the synthesis report generated by the Synopsys Design Compiler. It is seen that in this case there is no error.

5.3 SIMPLE load circuit

This unit consists of a combinational logic, whose inputs are single bits of data stored in a single bit register (flip-flop) and are available at the rising edge of the clock.. The difference

Generic delays of the model	Values from the report	Values from Backann2
OR2S_DEL	0.6150	0.6150
FN1_DEL	0.7700	0.7700
AND_DEL	0.6250	0.6250
SUM2_DEL	2.3750	2.3750
ENBL_DEL	0.9850	0.9850
MUPX_DEL	1.2850	1.2850
MUPX2_DEL	1.3200	1.3200
ENABLE_DEL	1.8200	1.8200
SUM1_DEL	2.2650	2.2650
CARRY1_DEL	1.4050	1.4050
CARRY2_DEL	1.4050	1.4050

Figure 15 Comparison of delays for ADDER-MUX circuit

between the processing of this sequential circuit and a purely combinational circuit, is that since registers are involved, a clock must be specified in the log_file.

The Process Model Graph is shown in Figure 16. The paths and the calculations involved are shown below.

TP1 = REG1B_DEL+INV_DELB+AND_DEL1
 TP2 = REG1A_DEL+INV_DELA+AND_DEL1

TP1 = 2.41
 TP2 = 2.41

AND_DEL1 = 0.290000
 INV_DELB = 0.150000
 INV_DELA = 0.150000
 REG1B_DEL = 0.680000
 REG1A_DEL = 0.680000

TP1_newx = REG1B_DEL + INV_DELB +
 AND_DEL1 ;
 REG1B_DEL_new = REG1B_DEL/TP1_newx*TP1
 REG1B_DEL = REG1B_DEL_new
 INV_DELB_new = INV_DELB/TP1_newx*TP1

$$\begin{aligned}
INV_DELB &= INV_DELB_new \\
AND_DEL1_new &= AND_DEL1/TP1_new \times TP1 \\
AND_DEL1 &= AND_DEL1_new
\end{aligned}$$

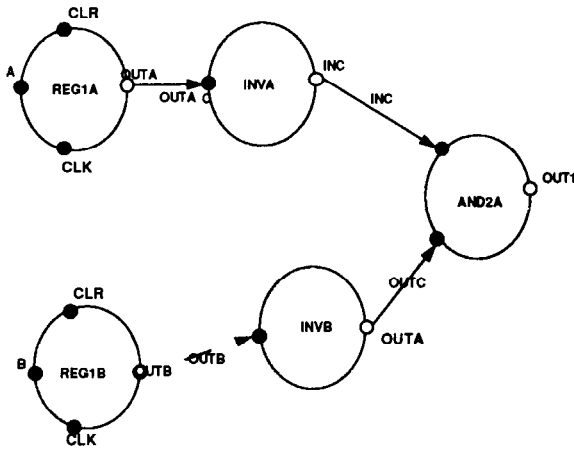


Figure 16 Process Model Graph of SIMPLE

$$\begin{aligned}
TP2 &= TP2_AND_DEL1 \\
TP2_newx &= REG1A_DEL + INV_DELA ; \\
REG1A_DEL_new &= REG1A_DEL / TP2_newx \times TP2 \\
REG1A_DEL &= REG1A_DEL_new \\
INV_DELA_new &= INV_DELA / TP2_newx \times TP2 \\
INV_DELA &= INV_DELA_new
\end{aligned}$$

The delay values calculated by MATLAB are shown below.

$$\begin{aligned}
AND_DEL1 &= 0.6240 \\
REG1A_DEL &= 1.4632 \\
REG1B_DEL &= 1.4632 \\
INV_DELA &= 0.3288 \\
INV_DELA &= 0.3228
\end{aligned}$$

The delay values obtained are compared with the delays obtained from the Synopsys Design compiler and are shown in Figure 17.

From Figure 17 it is seen that the average error percentage is 2.69% and the individual error percentage for AND_DEL, INV_DEL and REG1_DEL are 0.16%, 4.12 % and 0.8% respectively.

The example shown above can be extended to multiple registers and registers at the output.

Generic delays of the model	Values from the report	Values from Backann2
AND_DEL	0.6250	0.6240
INV_DELB	0.3100	0.3228
INVA_DEL	0.3100	0.3228
REG1B_DEL	1.4750	1.4632
REG1A_DEL	1.4750	1.4632

Figure 17 Comparison of Delays SIMPLE

5.4 Summary and Interpretation of Results

Three classes of circuits have been discussed in this chapter and the generic delay values for each model were found and compared with those obtained from the Synopsys Design Compiler. The following conclusions can be reached by interpretation of the above results :

1. The accuracy of backann2 does not depend on the size of the circuit but on the way each of the processes are connected.
2. When in the model there is at least one process whose inputs and outputs are the primary inputs and the primary outputs, the generic delay calculated is more accurate than when there is no such process.
3. The delay values backannotated are 100 % error free when there is one or more process which has a path from input to output and these processes when connected to other processes in the model, lead to equations such that these delays can be calculated without using the proportionality method. This is shown in the ADDER-MUX circuit.
4. The worst case error obtained in the results is close to 5%. It is seen that it occurs in models which there is no path in the circuit which has its primary input and primary outputs in the same process.

6. Conclusions

This paper has presented a back-annotation tool - Backann2. The results presented here and those obtained to date show that the tool achieves its purpose with reasonable accuracy. A tool of this sort is an important aid in the development of behavioral models with accurate timing. This tool thus helps quicken the design cycle from concept to silicon.

References

- [1] D. Giles, C. Berking, K. Wacks, "Integrated Functional/Structural Timing for Digital Simulation," *IEEE Test Conference*, pp.153-160, April 1982.
- [2] N.D.Dutt,D.D.Gajski,"Design Controlled Behavioral Synthesis," *26th ACM/IEEE Design Automation Conference*, 1989, pp754-757
- [3]N.D.Dutt, T.Hadley, D.D.Gajski, "An Intermediate Representation for Behavioral Synthesis," *27th ACM/IEEE Design Automation Conference*,1990,pp-14-19
- [4] J. R. Armstrong and F. G. Gray, *Structured Logic Design with VHDL*, Prentice Hall, 1993.
- [5] A. Gadagkar and J.R.Armstrong, "Timing Distribution in VHDL behavioral Models," *Proceedings of ICCAD 1992*, pp. 82-89.
- [6] A. Gadagkar, "Timing Distribution in VHDL Behavioral Models," Master's Thesis, Virginia Polytechnic Institute and State University, 1992.
- [7] S. Narayanaswamy, "Development of VHDL behavioral Models with Back Annotated timing," Master's Thesis, Virginia Polytechnic Institute and State University, 1993.
- [8] B. Singh, "A Parametrized CAD tool for VHDL Model Development with X-Windows," Master's Thesis, Virginia Polytechnic Institute and State University, 1990.
- [9] P. A. Wright, "Rapid Development of VHDL Behavioral Models," Master's Thesis, Virginia Polytechnic Institute and State University, 1992.
- [10] CAD Language Systems, *VHDL Analyzer Designer's Manual*, April 1993.
- [11] CAD Language Systems, *Design Library System*, April 1991.
- [12] CAD Language Systems, *DLS Application Development : The Software Procedural Interface*, March 1993.
- [13] Synopsys Inc., *Design Compiler Reference Manual*, Dec 1992.
- [14] S. R. Rao, "A Hierarchical Approach to Effective Test Generation for VHDL Behavioral Models," Master's Thesis, Virginia Polytechnic Institute and State University, 1992.
- [15] John F. Wakerly, *Digital Design Principles and Practices*, Prentice Hall, 1990
- [16] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, New York : North Holland, 1976.